Principles of Software Construction: Objects, Design, and Concurrency

Part 2: Design for large-scale reuse

API design (and some libraries and frameworks…)

Charlie Garrod **Bogdan Vasilescu**

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Homework 4c due tonight
  - Remember: up to 75% of lost points on 4a back
- Homework 5 coming soon
  - Team sign-up deadline March 27th
- Midterm exam in class next week Thursday (March 29th)
  - Review session next week Wednesday 5-7pm Margaret Morrison A14
- Optional reading for today
  - Effective Java, Items 52 (overloading) and 53 (variable arity methods)

https://commons.wikimedia.org/wiki/File:1_carcassonne_aerial_2016.jpg

Key concepts from last Tuesday

Motivation to create a public API

- Good APIs are a great asset
  - Distributed development among many teams
    - Incremental, non-linear software development
    - Facilitates communication
  - Long-term buy-in from clients & customers
    - Users invest heavily: acquiring, writing, learning
    - Cost to **stop** using an API can be prohibitive
    - Successful public APIs capture users

- Poor APIs are a great liability
  - Lost productivity from your software developers
  - Wasted customer support resources
  - Lack of buy-in from clients & customers

An API design process

- Define the scope of the API
  - Collect use-case stories, define requirements
  - Be skeptical
    - Distinguish true requirements from so-called solutions
    - **"When in doubt, leave it out."**
- Draft a specification, gather feedback, revise, and repeat
  - Keep it simple, short
  - Keep an issues list
- Code early, code often
  - **Write *client code* before you implement the API**

Josh Bloch's experiences designing the Collections framework

Conclusion

- **It takes a lot of work to make something that appears obvious**
  - Coherent, unified vision
  - Willingness to listen to others
  - Flexibility to accept change
  - Tenacity to resist change
  - Good documentation!
- **It's worth the effort!**
  - A solid foundation can last two+ decades

Learning goals for today

- Understand and be able to discuss the similarities and differences between API design and regular software design
  - Relationship between libraries, frameworks, and API design
  - Information hiding as a key design principle
- Acknowledge, and plan for failures as a fundamental limitation of a design process
- Given a problem domain with use cases, be able to plan a coherent design process for an API for those use cases
  - "Rule of Threes"

Outline

- The Process of API Design
- Key design principle: Information hiding
- Concrete advice for user-centered design

Key design principle: Information hiding

- "When in doubt, leave it out."

Documenting an API

- APIs should be self-documenting
  - Good names drive good design
- Document religiously anyway
  - All public classes
  - All public methods
  - All public fields
  - All method parameters
  - Explicitly write behavioral specifications
- Documentation is integral to the design and development process
- Do not document implementation details

Key design principle: Information hiding (2)

- Minimize the accessibility of classes, fields, and methods
  - You can add features, but never remove or change the behavioral contract for an existing feature

Which one do you prefer?

```
public class Point {
    public double x;
    public double y;
}
```

vs.

```
public class Point {
    private double x;
    private double y;
    public double getX() { /* … */ }
    public double getY() { /* … */ }
}
```

Key design principle: Information hiding (3)

- Use accessor methods, not public fields
  - Consider:

```
public class Point {
    public double x;
    public double y;
}
```

vs.

```
public class Point {
    private double x;
    private double y;
    public double getX() { /* … */ }
    public double getY() { /* … */ }
}
```

Key design principle: Information hiding (4)

- Prefer interfaces over (abstract) classes
  - Interfaces provide greater flexibility, avoid needless implementation details
  - Consider:

```
public interface Point {
    public double getX();
    public double getY();
}

public class PolarPoint() implements Point {
    private double r;       // Distance from origin.
    private double theta;   // Angle.
    public double getX() { return r*Math.cos(theta); }
    public double getY() { return r*Math.sin(theta); }
}
```

Which one do you prefer?

```
public class Rectangle {
  public Rectangle(Point e, Point f) …
}
```

vs.

```
public class Rectangle {
  public Rectangle(PolarPoint e, PolarPoint f) …
}
```

Still…

```
public class Rectangle {
 public Rectangle(Point e, Point f) …
}
…
Point p1 = new PolarPoint(…);
Point p2 = new PolarPoint(…);
Rectangle r = new Rectangle(p1, p2);
```

Still…

```
public class Rectangle {
 public Rectangle(Point e, Point f) …
}
…
Point p1 = new PolarPoint(…);
Point p2 = new PolarPoint(…);
Rectangle r = new Rectangle(p1, p2);
…
Point p3 = new PolarPoint(…);
Point p4 = new PolarPoint(…);
Rectangle r2 = new Rectangle(p3, p4);
…
```

Key design principle: Information hiding (5)

- Consider implementing a factory method instead of a constructor
  - Factory methods provide additional flexibility
    - Can be overridden
    - Can return instance of any subtype
      - Hides dynamic type of object
    - Can have a descriptive method name

Applying Information hiding: Factories

```
public class Rectangle {
 public Rectangle(Point e, Point f) …
}
…
Point p1 = PointFactory.Construct(…);
// new PolarPoint(…); inside
Point p2 = PointFactory.Construct(…);
// new PolarPoint(…); inside
Rectangle r = new Rectangle(p1, p2);
```

Key design principle: Information hiding (6)

- Prevent subtle leaks of implementation details
  - Documentation
  - Lack of documentation
  - Implementation-specific return types
  - Implementation-specific exceptions
  - Output formats
  - `implements Serializable`

Outline

- The Process of API Design
- Key design principle: Information hiding
- Concrete advice for user-centered design

Apply principles of user-centered design

- Other programmers are your users
- e.g., "Principles of Universal Design"
  - Equitable use
    - Design is useful and marketable to people with diverse abilities
  - **Flexibility in use**
    - Design accommodates a wide range of individual preferences
  - **Simple and intuitive use**
    - Use of the design is easy to understand
  - Perceptible information
    - Design communicates necessary information effectively to user
  - Tolerance for error
  - Low physical effort
  - Size and space for approach and use

Achieving flexibility in use while remaining simple and intuitive: Minimize conceptual weight

- API should be as small as possible but no smaller
  - **When in doubt, leave it out**
- Conceptual weight: How many concepts must a programmer learn to use
  - APIs should
- Good examp
  - java.uti
  - java.uti

`static <T> Collection<T>`	**`synchronizedCollection(Collection<T> c)`** Returns a synchronized (thread-safe) collection backed by the specified collection.
`static <T> List<T>`	**`synchronizedList(List<T> list)`** Returns a synchronized (thread-safe) list backed by the specified list.
`static <K,V> Map<K,V>`	**`synchronizedMap(Map<K,V> m)`** Returns a synchronized (thread-safe) map backed by the specified map.
`static <T> Set<T>`	**`synchronizedSet(Set<T> s)`** Returns a synchronized (thread-safe) set backed by the specified set.
`static <K,V> SortedMap<K,V>`	**`synchronizedSortedMap(SortedMap<K,V> m)`** Returns a synchronized (thread-safe) sorted map backed by the specified sorted map.
`static <T> SortedSet<T>`	**`synchronizedSortedSet(SortedSet<T> s)`** Returns a synchronized (thread-safe) sorted set backed by the specified sorted set.
`static <T> Collection<T>`	**`unmodifiableCollection(Collection<? extends T> c)`** Returns an unmodifiable view of the specified collection.
`static <T> List<T>`	**`unmodifiableList(List<? extends T> list)`** Returns an unmodifiable view of the specified list.
`static <K,V> Map<K,V>`	**`unmodifiableMap(Map<? extends K,? extends V> m)`** Returns an unmodifiable view of the specified map.
`static <T> Set<T>`	**`unmodifiableSet(Set<? extends T> s)`** Returns an unmodifiable view of the specified set.
`static <K,V> SortedMap<K,V>`	**`unmodifiableSortedMap(SortedMap<K,? extends V> m)`** Returns an unmodifiable view of the specified sorted map.
`static <T> SortedSet<T>`	**`unmodifiableSortedSet(SortedSet<T> s)`** Returns an unmodifiable view of the specified sorted set.

What's wrong here?

```
public class Thread implements Runnable {
   // Tests whether current thread has been interrupted.
   // Clears the interrupted status of current thread.
   public static boolean interrupted();
 }
```

Unintuitive behavior: side effects

- User of API should not be surprised by behavior, aka “the principle of least astonishment”
  - It's worth extra implementation effort
  - It's even worth reduced performance

```
public class Thread implements Runnable {
   // Tests whether current thread has been interrupted.
   // Clears the interrupted status of current thread.
   public static boolean interrupted();
}
```

Good names drive good design

- Do what you say you do:
  - "Don't violate the Principle of Least Astonishment"

Discuss these names:

- `get_x()` vs. `getX()` vs. `x()`
- `timer` vs. `Timer`
- `HTTPServlet` vs `HttpServlet`
- `isEnabled()` vs. `enabled()`
- `computeX()` vs. `generateX()`
- `deleteX()` vs. `removeX()`

Good names drive good design (2)

- Follow language- and platform-dependent conventions
  - Typographical:
    - `get_x()` vs. `getX()` vs. `x()`
    - `timer` vs. `Timer`, `HTTPServlet` vs `HttpServlet`
    - `edu.cmu.cs.cs214`
  - Grammatical (next slide):

Good names drive good design (3)

- Nouns for classes
  - `BigInteger`, `PriorityQueue`
- Nouns or adjectives for interfaces
  - `Collection`, `Comparable`
- Nouns, linking verbs or prepositions for non-mutative methods
  - `size`, `isEmpty`, `plus`
- Action verbs for mutative methods
  - `put`, `add`, `clear`

Good names drive good design (4)

- Use clear, specific naming conventions
  - `getX()` and `setX()` for simple accessors and mutators
  - `isX()` for simple boolean accessors
  - `computeX()` for methods that perform computation
  - `createX()` or `newInstance()` for factory methods
  - `toX()` for methods that convert the type of an object
  - `asX()` for wrapper of the underlying object

Good names drive good design (5)

- Be consistent
  - `computeX()` vs. `generateX()`?
  - `deleteX()` vs. `removeX()`?
- Avoid cryptic abbreviations
  - Good: `Font`, `Set`, `PrivateKey`, `Lock`, `ThreadFactory`, `TimeUnit`, `Future<T>`
  - Bad: `DynAnyFactoryOperations`, `_BindingIteratorImplBase`, `ENCODING_CDR_ENCAPS`, `OMGVMCID`

Do not violate Liskov's behavioral subtyping rules

- Use inheritance only for true subtypes
- Examples:

```
1)
class Stack extends Vector …
```

```
2)
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
    public Object put(Object key, Object value);
    …
}
```

Favor composition over inheritance

```
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
    public Object put(Object key, Object value);
    …
}

public class Properties {
    private final HashTable data = new HashTable();
    public String put(String key, String value) {
        data.put(key, value);
    }
    …
}
```

Minimize mutability

- Classes should be immutable unless there's a good reason to do otherwise
  - Advantages: simple, thread-safe, reusable
    - See java.lang.String
  - Disadvantage: separate object for each value

Example of bad API design decision: Java's AWT

- `Component.getSize()` returns `Dimension`
- `Dimension` is mutable
- Each `getSize` call must allocate `Dimension`
- Causes millions of needless object allocations
- Alternative added in Java1.2 but old client code still slow: `getX()`, `getY()`
- Document mutability
  - Carefully describe state space
  - Make clear when it's legal to call which method

On a piece of paper (in groups of 2-3)

1. Write your Andrew IDs.
2. Argue that a Carcassonne board implementation should be *mutable*. Explicitly include design goals and design principles in your rationale, where possible.
3. Argue that a Carcassonne board implementation should be *immutable*. Explicitly include …

https://commons.wikimedia.org/wiki/File:1_carcassonne_aerial_2016.jpg

Overload method names judiciously

- Avoid ambiguous overloads for subtypes
  - Recall the subtleties of method dispatch:

```
public class Point() {
        private int x;
        private int y;
        public boolean equals(Point p) {
            return x == p.x && y == p.y;
        }
    }
```

- If you must be ambiguous, implement consistent behavior

```
public class TreeSet implements SortedSet {
  public TreeSet(Collection c);  // Ignores order.
  public TreeSet(SortedSet s);   // Respects order.
}
```

Use appropriate parameter & return types

- Favor interface types over classes for input
  - Provides flexibility, performance
- Use most specific possible input parameter type
  - Moves error from runtime to compile time
- Don't use `String` if a better type exists
  - Strings are cumbersome, error-prone, and slow
- Don't use floating point for monetary values
  - Binary floating point causes inexact results!
- Use `double` (64 bits) rather than `float` (32 bits)
  - Precision loss is real, performance loss negligible

Use consistent parameter ordering

- An egregious example from C:

```
char* strncpy(char* dest, char* src,  size_t n);
void    bcopy(void* src,  void* dest, size_t n);
```

- Some good examples:

  `java.util.Collections`: first parameter is always the collection to be modified or queried

  `java.util.concurrent`: time is always specified as `long delay, TimeUnit unit`

Avoid long lists of parameters

- Especially with repeated parameters of the same type

```
HWND CreateWindow(LPCTSTR lpClassName, LPCTSTR lpWindowName,
  DWORD dwStyle, int x, int y, int nWidth, int nHeight,
  HWND hWndParent, HMENU hMenu, HINSTANCE hInstance,
  LPVOID lpParam);
```

- Long lists of identically typed params harmful
  - Programmers transpose parameters by mistake
  - Programs still compile and run, but misbehave!
- Three or fewer parameters is ideal
- Techniques for shortening parameter lists
  - Break up method
  - Create helper class to hold parameters
  - *Builder* Pattern

The *Effective Java*-style builder pattern


```
NutritionFacts twoLiterDietCoke = new NutritionFacts.Builder(
    "Diet Coke", 240, 6).sodium(1).build();

public class NutritionFacts {
    public static class Builder {
        public Builder(String name, int servingSize,
                       int servingsPerContainer) { ... }
        public Builder totalFat(int val)     { totalFat = val; …}
        public Builder saturatedFat(int val) { satFat = val; …}
        public Builder transFat(int val)     { transFat = val; …}
        public Builder cholesterol(int val)  { cholesterol = val; …
        ... // 15 more setters

        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }
    private NutritionFacts(Builder builder) { ... }
}
```

What's wrong here?

```
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
  public Object put(Object key, Object value);

  // Throws ClassCastException if this instance
  // contains any keys or values that are not Strings
  public void save(OutputStream out, String comments);
}
```

Fail fast

- Report errors as soon as they are detectable
  - Check preconditions at the beginning of each method
  - Avoid dynamic type casts, run-time type-checking

```
// A Properties instance maps Strings to Strings
public class Properties extends HashTable {
  public Object put(Object key, Object value);

  // Throws ClassCastException if this instance
  // contains any keys or values that are not Strings
  public void save(OutputStream out, String comments);
}
```

Avoid behavior that demands special processing

- Do not return `null` to indicate an empty value
  - e.g., Use an empty `Collection` or array instead
- Do not return `null` to indicate an error
  - Use an exception instead
- Use `Optional<T>` judiciously

Avoid return values that demand exceptional processing

- Return zero-length array or empty collection, not null

```
package java.awt.image;
public interface BufferedImageOp {
  // Returns the rendering hints for this operation,
  // or null if no hints have been set.
  public RenderingHints getRenderingHints();
}
```

- Do not return a `String` if a better type exists

Throw exceptions only for exceptional behavior

- Do not force client to use exceptions for control flow:

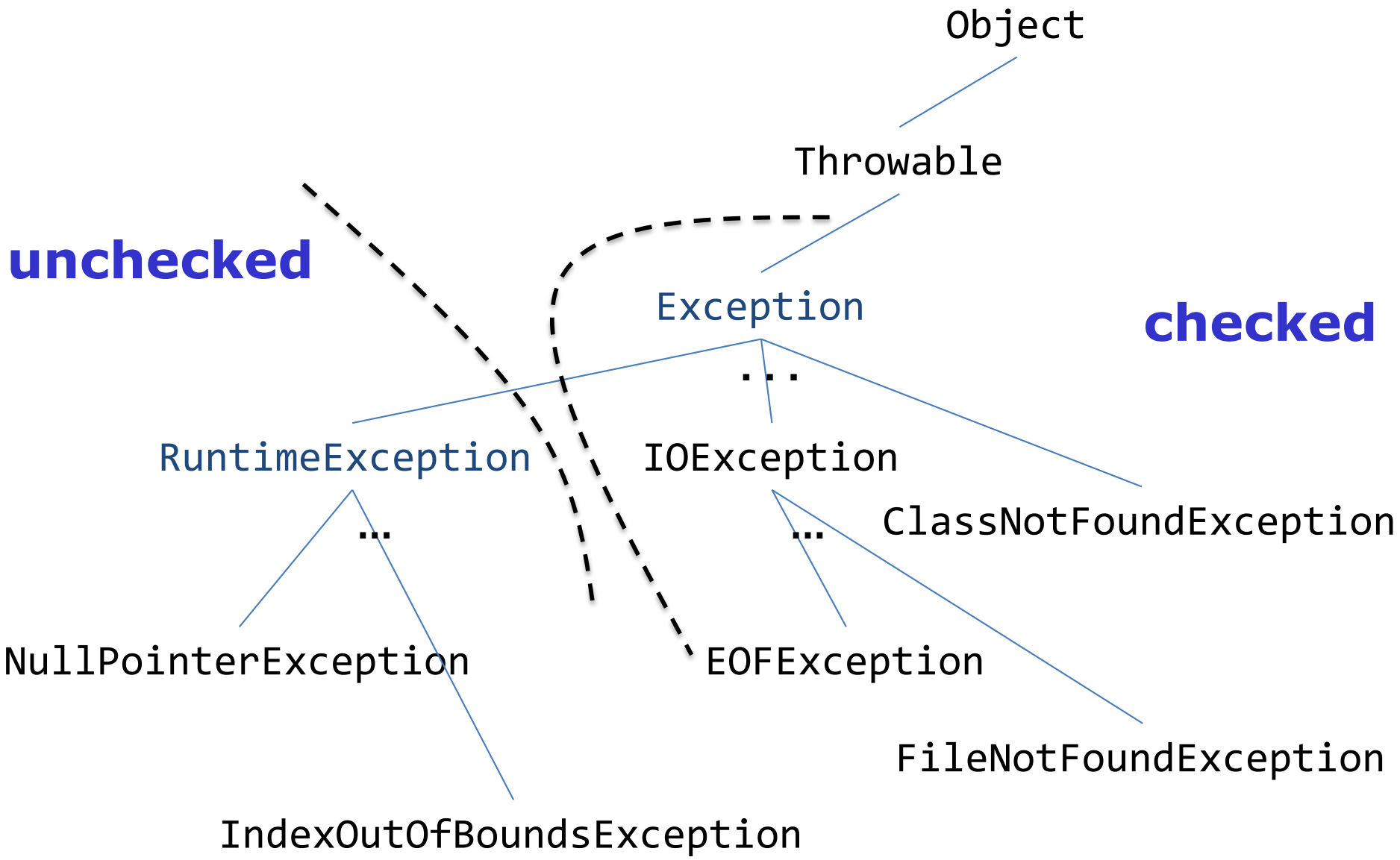
```
private byte[] a = new byte[CHUNK_SIZE];

void processBuffer (ByteBuffer buf) {
  try {
    while (true) {
      buf.get(a);
      processBytes(a, CHUNK_SIZE);
    }
  } catch (BufferUnderflowException e) {
    int remaining = buf.remaining();
    buf.get(a, 0, remaining);
    processBytes(a, remaining);
  }
}
```

- Conversely, don't fail silently:

```
ThreadGroup.enumerate(Thread[] list)
```

Context: The exception hierarchy in Java



Avoid checked exceptions, if possible

- Overuse of checked exceptions causes boilerplate code:

```
try {
    Foo f = (Foo) g.clone();
} catch (CloneNotSupportedException e) {
    // This exception can't happen if Foo is Cloneable
    throw new AssertionError(e);
}
```

Don't make the client do anything the module could do

- Carelessly written APIs force clients to write boilerplate code:

```
import org.w3c.dom.*;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

/** DOM code to write an XML document to a specified output stream. */
static final void writeDoc(Document doc, OutputStream out)throws IOException{
  try {
    Transformer t = TransformerFactory.newInstance().newTransformer();
    t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, doc.getDoctype().getSystemId());
    t.transform(new DOMSource(doc), new StreamResult(out)); // Does actual writing
  } catch(TransformerException e) {
    throw new AssertionError(e);  // Can’t happen!
  }
}
```

Don't let your output become your de facto API

- Document the fact that output formats may evolve in the future
- Provide programmatic access to all data available in string form

```
org.omg.CORBA.MARSHAL: com.ibm.ws.pmi.server.DataDescriptor; IllegalAccessException  minor code: 4942F23E  comp
        at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:199)
        at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
        at com.ibm.rmi.io.ValueHandlerImpl.read_Array(ValueHandlerImpl.java:625)
        at com.ibm.rmi.io.ValueHandlerImpl.readValueInternal(ValueHandlerImpl.java:273)
        at com.ibm.rmi.io.ValueHandlerImpl.readValue(ValueHandlerImpl.java:189)
        at com.ibm.rmi.iiop.CDRInputStream.read_value(CDRInputStream.java:1429)
        at com.ibm.ejs.sm.beans._EJSRemoteStatelessPmiService_Tie._invoke(_EJSRemoteStatelessPmiService_Tie.ja
        at com.ibm.CORBA.iiop.ExtendedServerDelegate.dispatch(ExtendedServerDelegate.java:515)
        at com.ibm.CORBA.iiop.ORB.process(ORB.java:2377)
        at com.ibm.CORBA.iiop.OrbWorker.run(OrbWorker.java:186)
        at com.ibm.ejs.oa.pool.ThreadPool$PooledWorker.run(ThreadPool.java:104)
        at com.ibm.ws.util.CachedThread.run(ThreadPool.java:137)
```

Don't let your output become your de facto API

- Document the fact that output formats may evolve in the future
- Provide programmatic access to all data available in string form

```
public class Throwable {
  public void printStackTrace(PrintStream s);
  public StackTraceElement[] getStackTrace(); // since 1.4
}

public final class StackTraceElement {
  public String  getFileName();
  public int     getLineNumber();
  public String  getClassName();
  public String  getMethodName();
  public boolean isNativeMethod();
}
```

Summary

- Accept the fact that you, and others, will make mistakes
  - Use your API as you design it
  - Get feedback from others
  - Hide information to give yourself maximum flexibility later
  - Design for inattentive, hurried users
  - Document religiously